浙江大学

本科实验报告

课程名称：操作系统

姓　　名：

学　　院：计算机科学与技术学院

　　系：计算机科学与技术系

专　　业：计算机科学与技术

学　　号：

指导教师：李环

2024 年 9 月 30 日

浙江大学操作系统实验报告

实验名称：RV64 内核引导与时钟中断处理

电子邮件地址：____手机：_____

实验地点：曹西 503 实验日期：2024 年 9 月 30 日

一、实验目的和要求

学习 RISC-V 汇编，编写 head.S 实现跳转到内核运行的第一个 C 函数；

学习 OpenSBI，理解 OpenSBI 在实验中所起到的作用，并调用 OpenSBI 提供的接口完成字符的输出；

学习 Makefile 相关知识，补充项目中的 Makefile 文件，来完成对整个工程的管理；

学习 RISC-V 的 trap 处理相关寄存器与指令，完成对 trap 处理的初始化；

理解 CPU 上下文切换机制，并正确实现上下文切换功能；

编写 trap 处理函数，完成对特定 trap 的处理；

调用 OpenSBI 提供的接口，完成对时钟中断事件的设置。

二、实验过程

（一）RV64 内核引导

1. 完善 makefile 脚本

阅读文档中关于 GNU Make 的章节，以及工程文件中的 Makefile 文件，根据注释学会 Makefile 的使用规则后，补充 lib/Makefile，使工程得以编译。

```
1.  C_SRC       = $(sort $(wildcard *.c))
2.  OBJ     = $(patsubst %.c,%.o,$(C_SRC))
```

```
3.
4.    all:$(OBJ)
5.
6.    %.o:%.c
7.     ${GCC} ${CFLAG} -c $<
8.    clean:
9.     $(shell rm *.o 2>/dev/null)
```

2. 编写 head.S

首先为即将运行的第一个 C 函数设置程序栈（栈的大小可以设置为 4KiB），并将该栈放置在 .bss.stack 段。接下来只需要通过跳转指令，跳转至 main.c 中的 start_kernel 函数即可。

```
1.        .extern start_kernel
2.        .section .text.entry
3.        .globl _start
4.    _start:
5.        la sp, boot_stack_top
6.        jal start_kernel #store the address of stack top and jump to
    start_kernel
7.
8.        .section .bss.stack
9.        .globl boot_stack
10.   boot_stack:
11.       .space 4096 # <-- change to your stack size
12.
13.       .globl boot_stack_top
14.   boot_stack_top:
```

3. 补充 sbi.c

OpenSBI 在 M 态，为 S 态提供了多种接口，比如字符串输入输出。因此我们需要实现调用 OpenSBI 接口的功能。给出函数定义如下：

```
1.    struct sbiret {
2.        uint64_t error;
3.        uint64_t value;
```

```
};

struct sbiret sbi_ecall(uint64_t eid, uint64_t fid,
                        uint64_t arg0, uint64_t arg1, uint64_t arg2,
                        uint64_t arg3, uint64_t arg4, uint64_t arg5);
```

sbi_ecall 函数中，需要完成以下内容：

将 eid（Extension ID）放入寄存器 a7 中，fid（Function ID）放入寄存器 a6 中，将 arg[0-5] 放入寄存器 a[0-5] 中。

使用 ecall 指令。ecall 之后系统会进入 M 模式，之后 OpenSBI 会完成相关操作。

OpenSBI 的返回结果会存放在寄存器 a0，a1 中，其中 a0 为 error code，a1 为返回值，我们用 sbiret 来接受这两个返回值。

```
#include "stdint.h"
#include "sbi.h"

struct sbiret sbi_ecall(uint64_t eid, uint64_t fid,
                        uint64_t arg0, uint64_t arg1, uint64_t arg2,
                        uint64_t arg3, uint64_t arg4, uint64_t arg5) {
    struct sbiret retval;
    uint64_t error, value;
    __asm__ volatile (
        "mv a7,%[eid]\n"
        "mv a6,%[fid]\n"
        "mv a5,%[arg5]\n"
        "mv a4,%[arg4]\n"
        "mv a3,%[arg3]\n"
        "mv a2,%[arg2]\n"
        "mv a1,%[arg1]\n"
        "mv a0,%[arg0]\n"
        "ecall\n"
        "mv %[error],a0\n"
        "mv %[value],a1\n"
        : [error] "=r"(error), [value] "=r"(value)
        : [eid] "r"(eid), [fid] "r"(fid),
          [arg5] "r"(arg5), [arg4] "r"(arg4),
          [arg3] "r"(arg3), [arg2] "r"(arg2),
          [arg1] "r"(arg1), [arg0] "r"(arg0)
```

```
            : "memory","a0","a1","a2","a3","a4","a5","a6","a7"
        );
        retval.error = error;
        retval.value = value;
        return retval;
    }

    struct sbiret sbi_debug_console_write_byte(uint8_t byte) {
        return sbi_ecall(0x4442434E, 0x2, byte, 0, 0, 0, 0, 0);
    }

    struct sbiret sbi_system_reset(uint32_t reset_type, uint32_t rese
t_reason) {
        return sbi_ecall(0x53525354, 0, reset_type, reset_reason, 0,
0, 0, 0);
    }
```

4. 修改 defs.h

补充 arch/riscv/include/defs.h 中的代码，完成 read_csr 的宏定义。

```
#ifndef __DEFS_H__
#define __DEFS_H__

#include "stdint.h"

#define csr_read(csr)                   \
  ({                                    \
    uint64_t __v;                       \
    asm volatile ("csrr " "%0, " #csr   \
                  :"=r" (__v);          \
                  :"memory");           \
    __v;                                \
  })

#define csr_write(csr, val)                                 \
  ({                                                        \
    uint64_t __v = (uint64_t)(val);                         \
    asm volatile("csrw " #csr ", %0" : : "r"(__v) : "memory"); \
  })

#endif
```

完成后，在工程代码根目录输入 make run，即可执行，正确输出 2024 ZJU Operating System 并正常退出。

（二）RV64 时钟中断处理

1. 修改 vmlinux.lds 和 head.S 的准备工作

修改后的 vmlinux.lds 如下：

```
/* 目标架构 */
OUTPUT_ARCH("riscv")

/* 程序入口 */
ENTRY(_start)

/* kernel 代码起始位置 */
BASE_ADDR = 0x80200000;

SECTIONS
{
    /* . 代表当前地址 */
    . = BASE_ADDR;

    /* 记录 kernel 代码的起始地址 */
```

```
16.         _skernel = .;
17.
18.         /* ALIGN(0x1000) 表示 4KiB 对齐 */
19.         /* _stext, _etext 分别记录了 text 段的起始与结束地址 */
20.         .text : ALIGN(0x1000) {
21.             _stext = .;
22.
23.             *(.text.init)
24.             *(.text.entry)
25.             *(.text .text.*)
26.
27.             _etext = .;
28.         }
29.
30.         .rodata : ALIGN(0x1000) {
31.             _srodata = .;
32.
33.             *(.rodata .rodata.*)
34.
35.             _erodata = .;
36.         }
37.
38.         .data : ALIGN(0x1000) {
39.             _sdata = .;
40.
41.             *(.data .data.*)
42.
43.             _edata = .;
44.         }
45.
46.         .bss : ALIGN(0x1000) {
47.             _sbss = .;
48.
49.             *(.bss.stack)
50.             *(.sbss .sbss.*)
51.             *(.bss .bss.*)
52.
53.             _ebss = .;
54.         }
55.
56.         /* 记录 kernel 代码的结束地址 */
57.         _ekernel = .;
58.     }
```

修改后的 head.S 如下：

```
15.       .extern start_kernel
16.       .section .text.init
17.       .globl _start
18.   _start:
19.       la sp, boot_stack_top
20.       jal start_kernel #store the address of stack top and jump to
    start_kernel
21.
22.       .section .bss.stack
23.       .globl boot_stack
24.   boot_stack:
25.       .space 4096 # <-- change to your stack size
26.
27.       .globl boot_stack_top
28.   boot_stack_top:
```

2. 开启 trap 处理

在运行 start_kernel 之前，我们要对上面提到的 CSR 进行初始化，初始化包括以下几个步骤：

（1）设置 stvec

将 _traps 所表示的地址写入 stvec

这里我们采用 Direct 模式，而 _traps 则是 trap 处理入口函数的基地址。

（2）开启时钟中断，将 sie[STIE] 置 1

（3）设置第一次时钟中断，参考 clock_set_next_event()中的逻辑用汇编实现

（4）开启 S 态下的中断响应，将 sstatus[SIE] 置 1

```
1.        .extern start_kernel
2.        .section .text.init
3.        .globl _start
4.    _start:
5.        la t0,_traps
6.        csrw stvec,t0 #stvec = _traps
7.
8.        csrr t0,sie
9.        ori t0,t0,0x20
10.       csrw sie,t0 #sie[STIE] = sie[5] = 1
```

```
11.
12.       andi a1,x0,0
13.       andi a2,x0,0
14.       andi a3,x0,0
15.       andi a4,x0,0
16.       andi a5,x0,0
17.       andi a6,x0,0
18.       andi a7,x0,0
19.       li t0,10000000
20.       rdtime a0
21.       add a0,a0,t0
22.       ecall #set first time interrupt
23.
24.       csrr t0,sstatus
25.       ori t0,t0,0x2
26.       csrw sstatus,t0 #sstatus[SIE] = sstatus[1] = 1
27.
28.       la sp, boot_stack_top
29.       jal start_kernel #store the address of stack top and jump to
    start_kernel
30.
31.       .section .bss.stack
32.       .globl boot_stack
33.   boot_stack:
34.       .space 4096 # <-- change to your stack size
35.
36.       .globl boot_stack_top
37.   boot_stack_top:
```

3. 实现上下文切换

（1）在 arch/riscv/kernel/ 目录下添加 entry.S 文件

（2）保存 CPU 的寄存器（上下文）到内存中（栈上）

（3）将 scause 和 sepc 中的值传入 trap 处理函数，我们将会在 trap_handler 中实现对 trap 的处理

（4）在完成对 trap 的处理之后，我们从内存中（栈上）恢复 CPU 的寄存器（上下文）

（5）从 trap 中返回

```
1.        .extern trap_handler
```

```
2.          .section .text.entry
3.          .align 2
4.          .globl _traps
5.      _traps:
6.          addi sp,sp,-33*8
7.          sd x0,0*8(sp)
8.          sd x1,1*8(sp)
9.          sd x2,2*8(sp)
10.         sd x3,3*8(sp)
11.         sd x4,4*8(sp)
12.         sd x5,5*8(sp)
13.         sd x6,6*8(sp)
14.         sd x7,7*8(sp)
15.         sd x8,8*8(sp)
16.         sd x9,9*8(sp)
17.         sd x10,10*8(sp)
18.         sd x11,11*8(sp)
19.         sd x12,12*8(sp)
20.         sd x13,13*8(sp)
21.         sd x14,14*8(sp)
22.         sd x15,15*8(sp)
23.         sd x16,16*8(sp)
24.         sd x17,17*8(sp)
25.         sd x18,18*8(sp)
26.         sd x19,19*8(sp)
27.         sd x20,20*8(sp)
28.         sd x21,21*8(sp)
29.         sd x22,22*8(sp)
30.         sd x23,23*8(sp)
31.         sd x24,24*8(sp)
32.         sd x25,25*8(sp)
33.         sd x26,26*8(sp)
34.         sd x27,27*8(sp)
35.         sd x28,28*8(sp)
36.         sd x29,29*8(sp)
37.         sd x30,30*8(sp)
38.         sd x31,31*8(sp)
39.         csrr t0,sepc
40.         sd t0,32*8(sp) #save registers and sepc
41.
42.         csrr a0,scause
43.         csrr a1,sepc
44.         jal trap_handler #trap handler
45.
```

```
46.         ld x0,0*8(sp)
47.         ld x1,1*8(sp)
48.         ld x2,2*8(sp)
49.         ld x3,3*8(sp)
50.         ld x4,4*8(sp)
51.         ld x5,5*8(sp)
52.         ld x6,6*8(sp)
53.         ld x7,7*8(sp)
54.         ld x8,8*8(sp)
55.         ld x9,9*8(sp)
56.         ld x10,10*8(sp)
57.         ld x11,11*8(sp)
58.         ld x12,12*8(sp)
59.         ld x13,13*8(sp)
60.         ld x14,14*8(sp)
61.         ld x15,15*8(sp)
62.         ld x16,16*8(sp)
63.         ld x17,17*8(sp)
64.         ld x18,18*8(sp)
65.         ld x19,19*8(sp)
66.         ld x20,20*8(sp)
67.         ld x21,21*8(sp)
68.         ld x22,22*8(sp)
69.         ld x23,23*8(sp)
70.         ld x24,24*8(sp)
71.         ld x25,25*8(sp)
72.         ld x26,26*8(sp)
73.         ld x27,27*8(sp)
74.         ld x28,28*8(sp)
75.         ld x29,29*8(sp)
76.         ld x30,30*8(sp)
77.         ld x31,31*8(sp)
78.         ld t0,32*8(sp)
79.         csrw sepc,t0
80.         addi sp,sp,33*8 #restore the value of the registers and sepc
81.         sret #return
```

4. 实现 trap 处理函数

（1）在 arch/riscv/kernel/ 目录下添加 trap.c 文件

（2）在 trap.c 中实现 trap 处理函数 trap_handler()，其接收的两个参数分别是 scause 和 sepc 两个寄存器中的值。

```c
#include "stdint.h"

void trap_handler(uint64_t scause, uint64_t sepc) {
    // 通过 `scause` 判断 trap 类型
    // 如果是 interrupt 判断是否是 timer interrupt
    if (scause >> 63 && scause % 8 == 5){
        // 如果是 timer interrupt 则打印输出相关信息，并通过 `clock_set_next_event()` 设置下一次时钟中断
        printk("[S] Supervisor Mode Timer Interrupt\n");
        // `clock_set_next_event()` 见 4.3.4 节
        clock_set_next_event();
    }
    // 其他 interrupt / exception 可以直接忽略，推荐打印出来供以后调试
}
```

5. 实现时钟中断相关函数

（1）在 arch/riscv/kernel/ 目录下添加 clock.c 文件

（2）在 clock.c 中实现 get_cycles()

使用 rdtime 汇编指令获得当前 time 寄存器中的值

（3）在 clock.c 中实现 clock_set_next_event()

调用 sbi_ecall，设置下一个时钟中断事件

```c
#include "stdint.h"

// QEMU 中时钟的频率是 10MHz，也就是 1 秒钟相当于 10000000 个时钟周期
uint64_t TIMECLOCK = 10000000;

uint64_t get_cycles() {
    // 编写内联汇编，使用 rdtime 获取 time 寄存器中（也就是 mtime 寄存器）的值并返回
    unsigned long t;
    __asm__ volatile(
        "rdtime %[t]"
        :[t] "=r" (t)
        : : "memory"
    );
    return t;
}

```

```
void clock_set_next_event() {
    // 下一次时钟中断的时间点
    uint64_t next = get_cycles() + TIMECLOCK;

    // 使用 sbi_set_timer 来完成对下一次时钟中断的设置
    sbi_ecall(0x00,0,next,0,0,0,0,0);
}
```

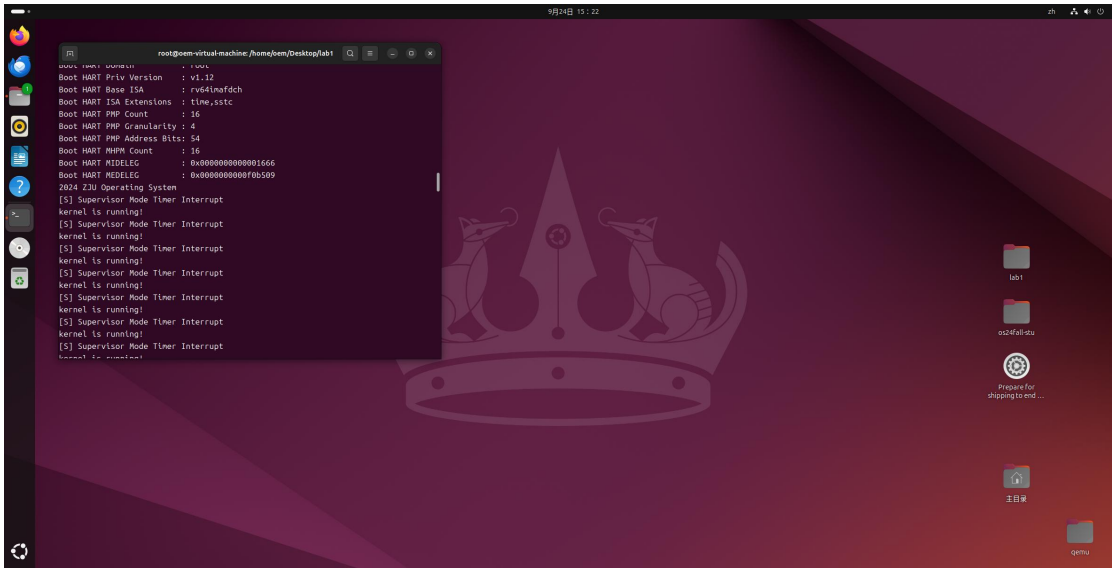
6. 修改 test 函数

为了让 kernel 持续运行，并使中断更容易观察，将 test.c 中的 test() 函数修改为如下内容：

```
#include "printk.h"

void test() {
    int i = 0;
    while (1) {
        if ((++i) % 100000000 == 0) {
            printk("kernel is running!\n");
            i = 0;
        }
    }
}
```

完成以上步骤后，即可编译运行。运行时，程序每隔一秒输出“[S] Supervisor Mode Timer Interrupt”和“kernel is running!”



三、讨论和心得

本次实验建立在 ubuntu 环境已经搭建好的基础上，做起来也没太多的困难，但是在实验过程中仍然遇到了几个问题。首先是 ubuntu 系统从 22.04 更新到 24.04 的问题，在输入指令更新的最后一步要求重启 ubuntu 系统，但是重启完成后整个系统陷入了黑屏的状态。当我把虚拟机关掉再重新打开 ubuntu 系统，虽然更新后的 ubuntu 能够进去但是时不时会卡死在系统界面。后来查阅了很多资料才发现需要在虚拟机设置-显示器-3D 图形那里关闭加速 3D 图形，这样就可以进去了，然后输入 sudo add-apt-repository ppa:oibaf/graphics-drivers sudo apt update && sudo apt upgrade 更新图形驱动。不得不说虚拟机的设置有时候还是十分反人类的，根本不会有人告诉我们更新的过程会发生什么。接下来就是一些代码的编写，例如 makefile 文件、内联汇编等等，在阅读资料后也是了解了其中的用法。还有一开始我不知道 sie[STIE]和 sstatus[SIE]是什么意思，查阅资料后才发现方括号内的字母代表下标，类似于宏定义。初次之外，别的代码并不是十分困难。

这次实验让我了解到了除 32 个基本寄存器之外的状态寄存器以及其作用，让我更深地了解操作系统中断和时钟的机制，同时也让我学会了 makefile 文件以及 RISC-V 架构下汇编代码的编写。

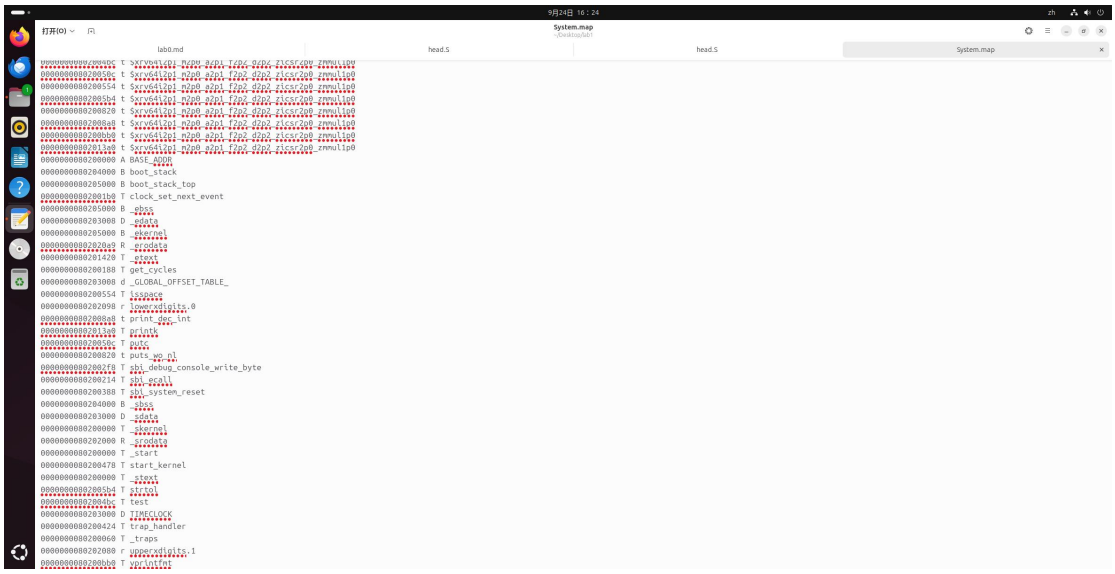
四、思考题

1.
RISC-V 的 calling convention 主要包括以下几点：
（1）参数传递：前 8 个参数通过 a0 到 a7 寄存器传递，超过 8 个的参数通过栈传递。
（2）返回值：返回值通常存放在 a0 寄存器。
（3）保存寄存器：调用者需要保存的寄存器有 s0 到 s11（callee-saved），而调用者使用的寄存器（t0 到 t6）在调用后不需要保存。

Caller Saved Register 和 Callee Saved Register 的区别在于：
Caller Saved Register（如 t 寄存器）由调用者负责保存和恢复。如果调用者需要保持这些寄存器的值，它必须在调用前保存它们，调用后恢复。
Callee Saved Register（如 s 寄存器）由被调用者负责保存和恢复。如果被调用的函数需要使

用这些寄存器，它必须在进入时保存当前的值，并在返回前恢复它们。这确保了调用者在函数调用后可以继续使用原来的值。

2.



3.

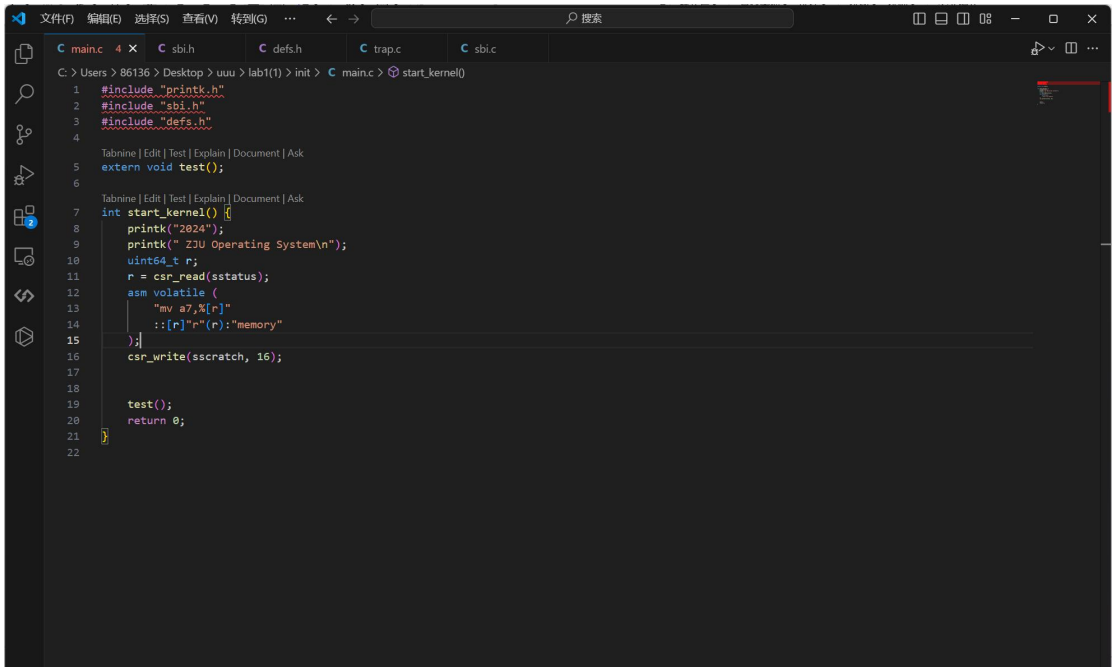
---

csrr rd, csr　　x[rd] = CSRs[csr]

读控制状态寄存器 *(Control and Status Register Read)*. 伪指令(Pesudoinstruction), RV32I and RV64I.

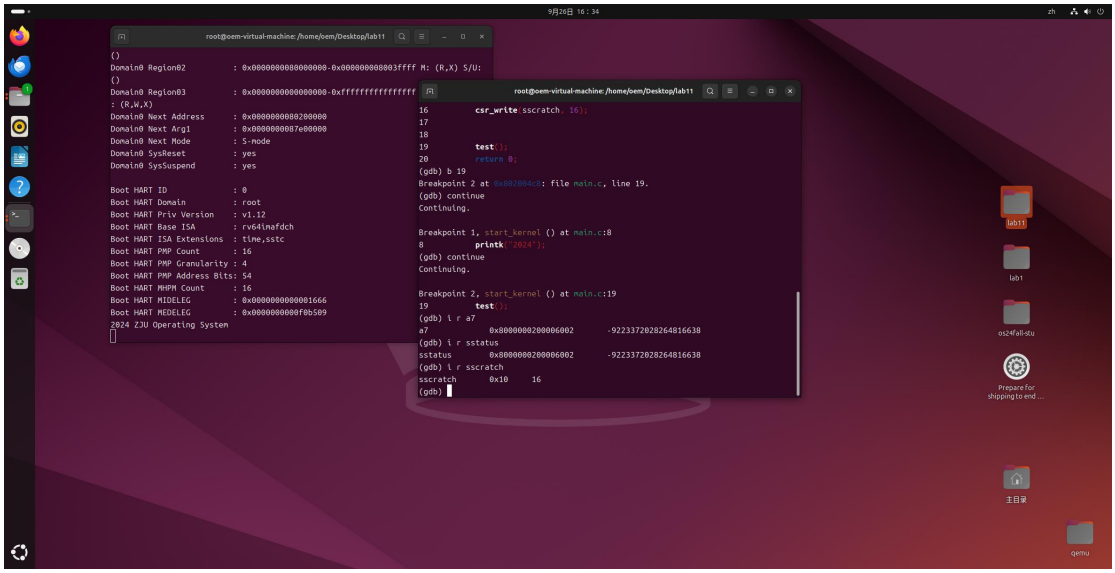
把控制状态寄存器 *csr* 的值写入 x[*rd*]，等同于 **csrrs** rd, csr, x0.

根据 RISC-V 手册，可得 csr_read 的含义是使用 csrr 指令将寄存器 sstatus 的值读取并传出。



修改后的 main.c 代码如上图，当使用 csr_read 将寄存器 sstatus 的值传入到变量 r 中时，r 又

将值传给了寄存器 a7。可以看到，a7 中的值和 sstatus 中的值一样，说明 csr_read 成功读取了 sstatus 的值。

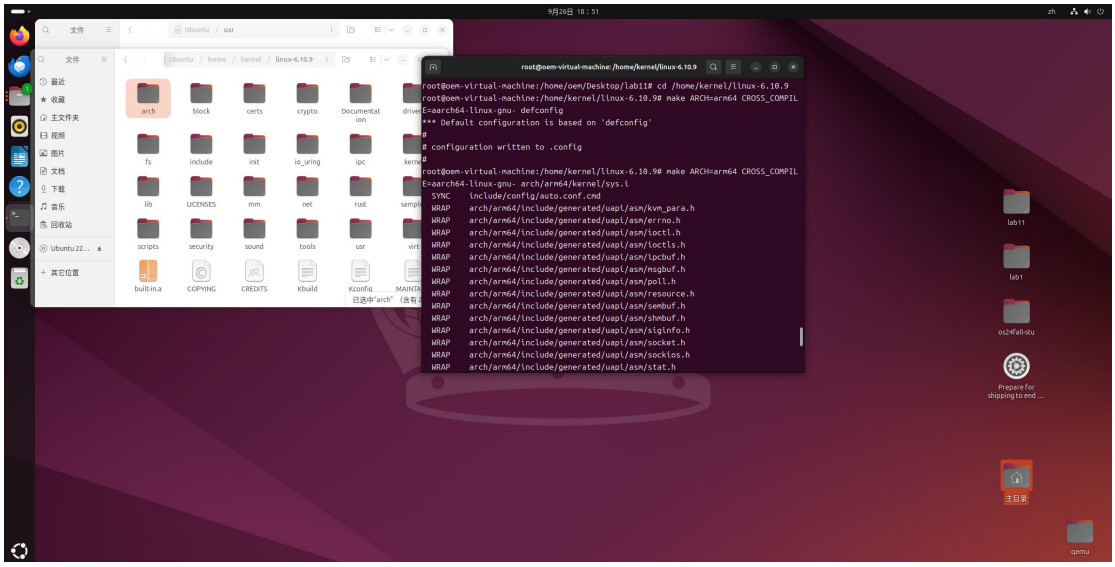


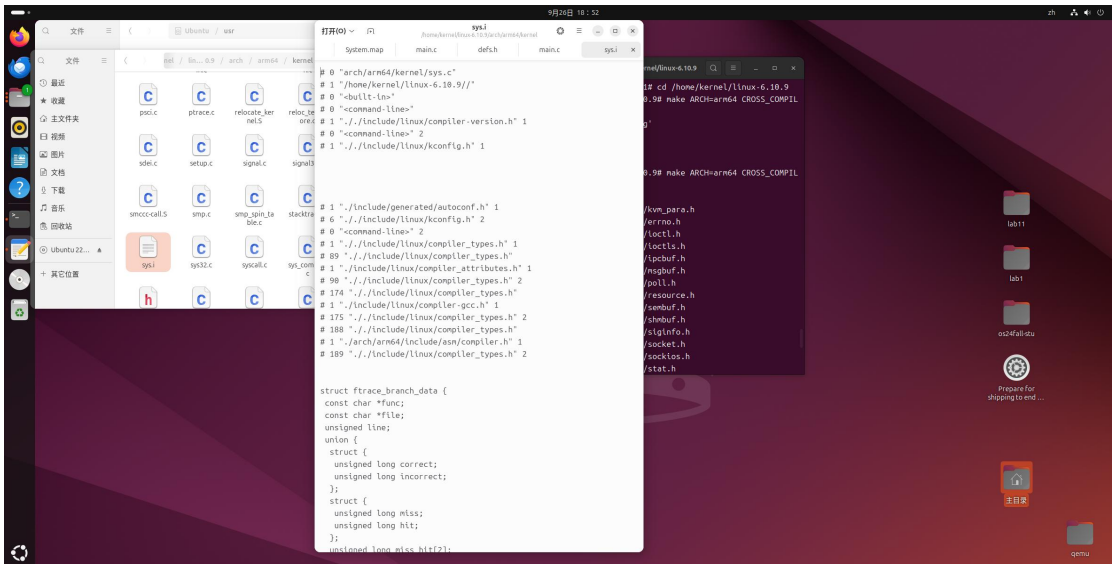
4.

如上图，在 main.c 中写入 csr_write(sscratch, 16)，通过 gdb 调试可以看到 sscratch 的值变成了 16。

5.

进入内核源码目录，输入“make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- defconfig ”“ make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- arch/arm64/kernel/sys.i”即可得到 sys.i 文件。





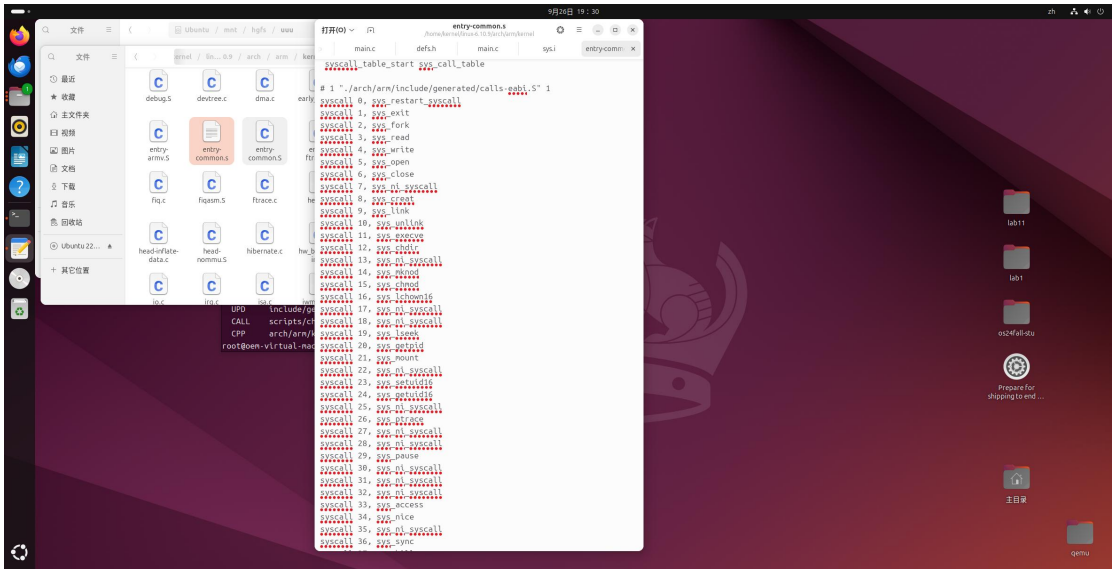
6.

ARM32:

在 arch/arm/kernel 目录下的 entry-common.S 找到如下代码段：

```
352     syscall_table_start sys_call_table
353     #ifdef CONFIG_AEABI
354     #include <calls-eabi.S>
355     #else
356     #include <calls-oabi.S>
357     #endif
358     syscall_table_end sys_call_table
359
```

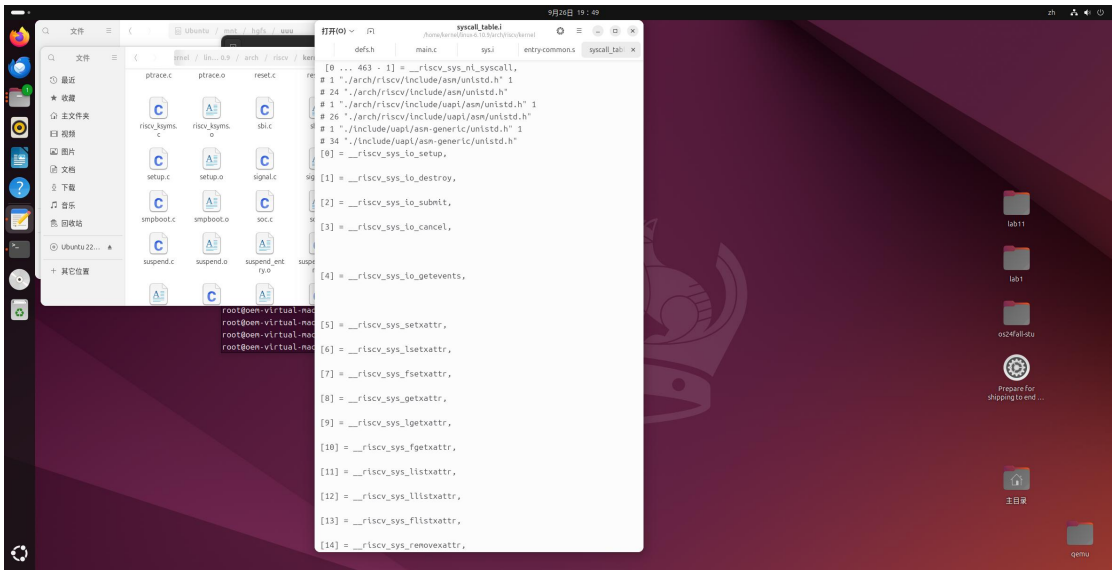
发现有两套系统调用表，分别是 calls-eabi.S 和 calls-oabi.S,在源码根目录输入命令 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- defconfig 和 make ARCH=arm CROSS_COMPILE=arm-linux-gnueabihf- arch/arm/kernel/entry-common.s ，生成 entry-common.s 文件，打开该文件即可看到宏展开后的系统调用表



RV32 & RV64：

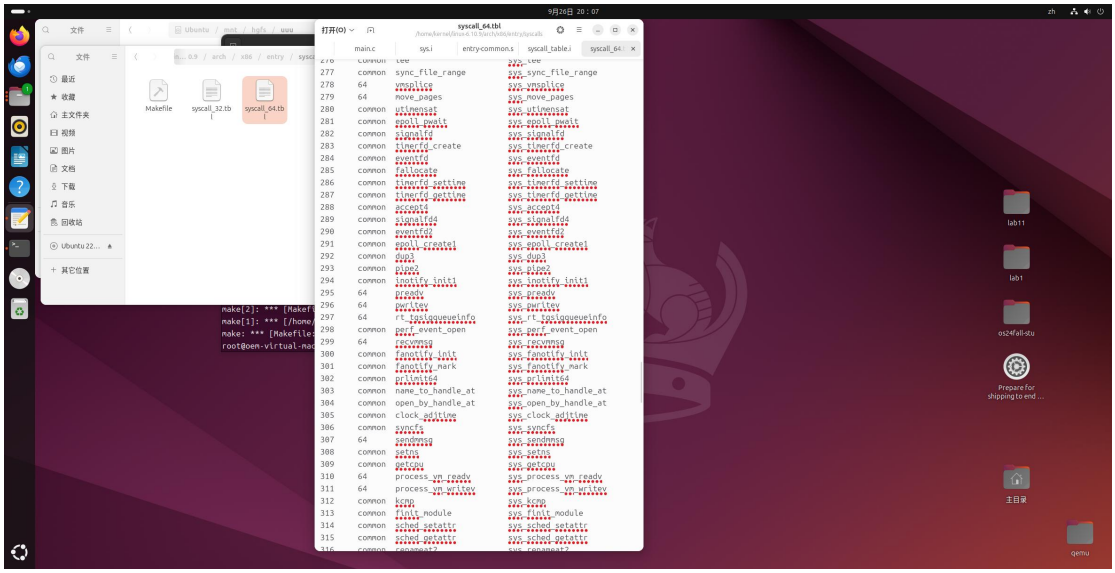
进入 arch/riscv/kernel 目录，发现系统调用表源文件 syscall_table.c

输入命令 make ARCH=riscv defconfig 和 make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- arch/riscv/kernel/syscall_table.i，生成 syscall_table.i 文件，打开该文件即可看到宏展开后的系统调用表



x86_64：

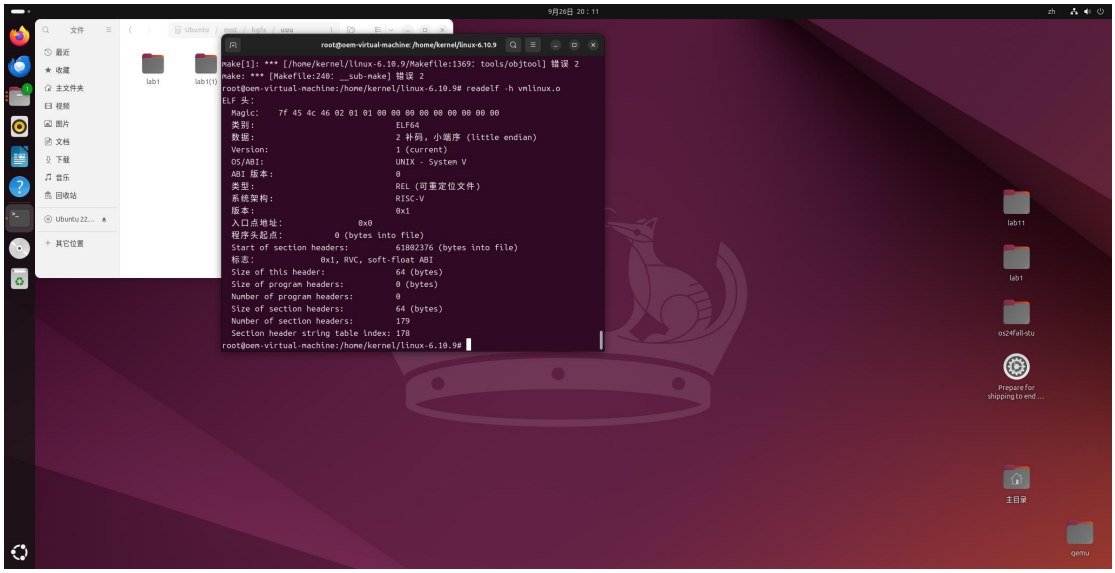
进入 arch/x86/entry/syscalls 目录，发现系统调用表源文件 syscall_64.tbl，打开该文件即可看到宏展开后的系统调用表



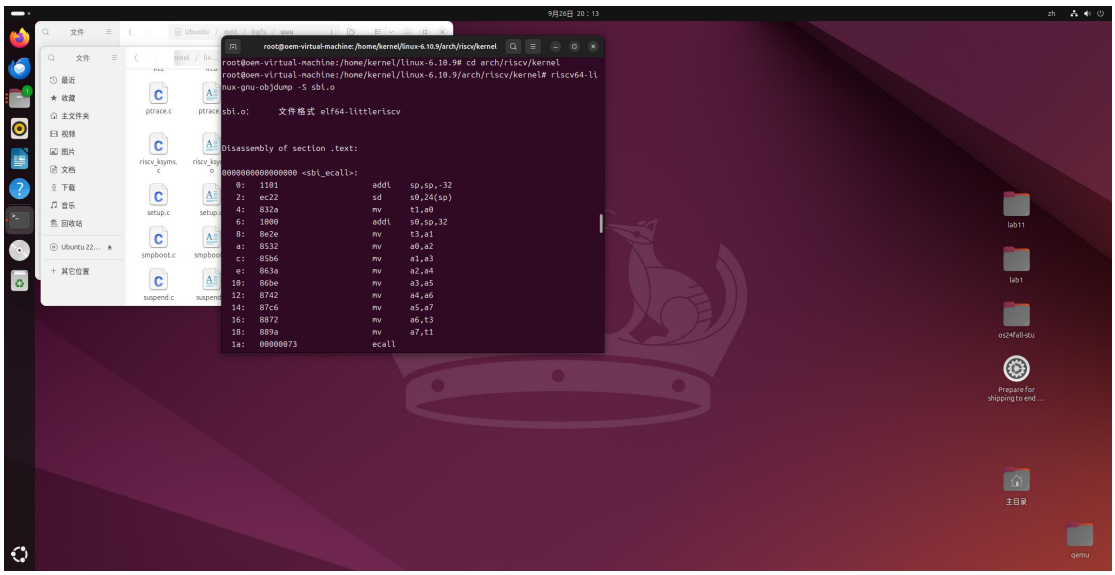
7.

ELF（Executable and Linkable Format）文件是一种用于存储可执行程序、目标代码、共享库和核心转储的文件格式。它是 Unix 和类 Unix 操作系统（如 Linux）中广泛使用的标准文件格式。

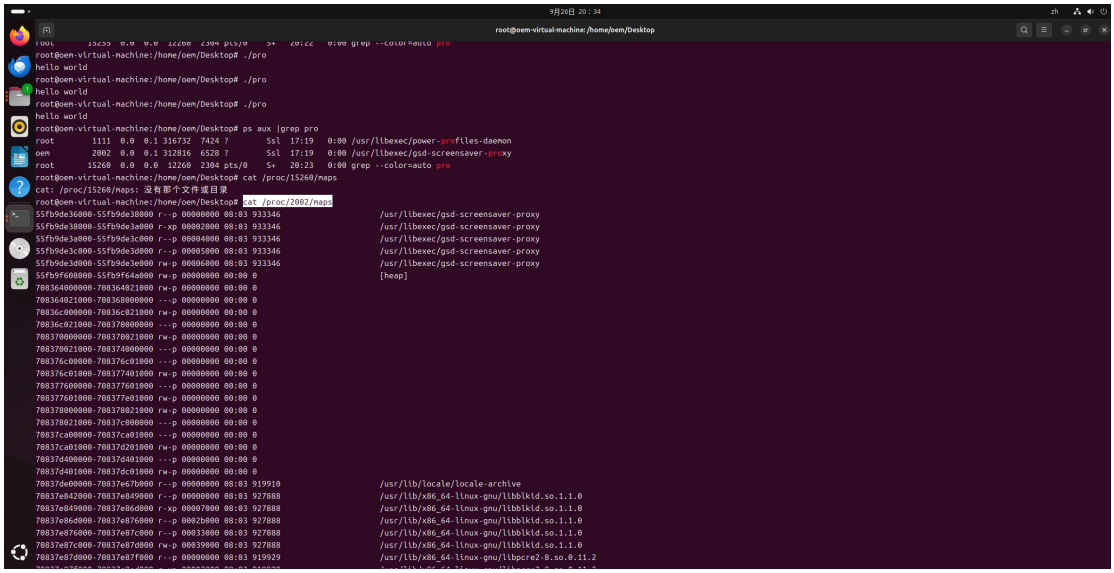
readelf：



objdump：



输入 ps aux |grep pro 可以找到运行的文件名包含 pro 的文件，这里我们找到 usr/libexec/gsd-screensaver-proxy，输入命令 cat /proc/2002/maps，结果如下：



8.

在 RISC-V Privileged Architecture Specification 中，MIDELEG 和 MEDELEG 寄存器用于控制中断和异常的委派，从而决定它们在不同特权模式下的处理方式。

将 MIDELEG 转换为二进制：0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0010 0010 0010

每一位代表一个中断（从 0 到 63）。

第 9 位：表示外部中断被委派给 S 模式。

第 5 位：表示定时器中断被委派给 S 模式。

第 1 位：表示软件中断被委派给 S 模式。

因此，MIDELEG 的设置表示这些中断将在 S 模式中被处理，而不是在 M 模式中处理。

将 MEDELEG 转换为二进制：0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 1011 0001 0000 1001

第 0 位：指令访问未对齐异常被委派给 S 模式。

第 3 位：断点异常被委派给 S 模式。

第 8 位：来自 U 模式的调用被委派给 S 模式。

第 12 位：指令页面异常被委派给 S 模式。

第 13 位：加载页面异常被委派给 S 模式。

第 15 位：store/AMO 页面异常被委派给 S 模式。

五、附录

无